Algorithmen und Datenstrukturen Zusammenfassung

Maximilian Wolf

16. Februar 2026

Inhaltsverzeichnis

1 Analysetools

1.1 Asymptotische Notation

Definition 1.1: $\mathcal{O}$-Notation

$\mathcal{O}(f(n))$ bezeichnet Klasse aller Funktionen g, die durch f eine **obere** Schranke haben

Definition 1.2: Ω-Notation

$\Omega(f(n))$ bezeichnet Klasse aller Funktionen g, die durch f eine **untere** Schranke haben

Definition 1.3: Θ-Notation

Mit $\Theta(f(n))$ bezeichnet man die Klasse $\mathcal{O}(f(n)) \cap \Omega(f(n))$, d.h. falls $g(n) \in \Theta(f(n))$, dann hat man das Wachstum der Funktion g bis auf einen konstanten Faktor und Terme niedriger Ordnung abgeschätzt.

1.2 Worst-Case und Average-Case

Definition 1.4: Worst-Case Komplexität

Laufzeit eines Algorithmus A bei der ungünstigsten Eingabe der Länge n. Es gilt: wc-time$_A(n) := max_{x:|x|=n}$time$_A(x)$

Definition 1.5: Average-Case Komplexität

Durchschnittliche Laufzeit eines Algorithmus A bei Gleichverteilung durch Zufallsvariable $T_{A,n}$ Es gilt: av-time$_A(n) := \mathbb{E}[T_{a,n}] = \frac{1}{|\{x:|x|=n\}|} \cdot \sum_{x:|x|=n}$ time$_A(x)$

1.3 Rekursionsgleichungen

1.3.1 Konstruktive Induktion

- Aufstellen einer Vermutung über die Komplexitätsklasse
- Führen eines Induktives Beweises über die Laufzeit
- Enthält oft noch unbekannte Parameter (z.B. $f(n) \leq cn$), die erst während des Induktionsbeweises durch Koeffizientenvergleich bestimmt werden

1.3.2 Iterationsmethode

- Rekursionsgleichung wird wiederholt, indem der rekursive Term immer wieder durch sich selbst (mit einem kleineren Argument) ersetzt wird

2 Sortier- und Selektionsalgorithmen

2.1 Sortierverfahren

2.1.1 Untere Schranke für vergleichsbasierte Sortierverfahren

Jeder auf paarweisen Element-Vergleichen basierende Sortieralgorithmus hat die worst-case Vergleichskomplexität $\geq n \log n - \Theta(n)$

2.1.2 MergeSort

```
1  IF |a| = 1 THEN
2    return
3  ELSE
4    Zerlege a in zwei Teile a' und a'' mit |a'| = |a''|
5    mergeSort(a')
6    mergeSort(a'')
7    Mische a' und a'' zu einem Array b
8    a := b
9    return
10
```

Dies ergibt die Rekursiongsleichung $V(n) = 2V(\frac{n}{2}) + n - 1$ mit $V(1) = 0$

2.1.3 QuickSort

```
1 IF rechts - links >= 1 THEN
2   Waehle ein Element x aus a[links..rechts], z.B. x = a [
     links]
3   Ordne a[links .. rechts] so um, dass alle Array-Elemente,
     die kleiner (groesser) als x sind, links (rechts) von x
     angeordnet werden.
4   Sei q die Array-Position, die x bei der Umordnung von a
     erhaelt.
5   quickSort(links, q - 1)
6   quickSort(q + 1, rechts)
7
```

- Typisches Divide-and-Conquer Verfahren
- Beste av-case aufzeit von $\Theta(n \log n)$

- wc-laufzeit von $\Theta(n^2)$ gerade der
- Im günstigsten Fall (jeder Pivot ist Median) gilt die Rekursionsgleichung $V(n) = n - 1 + 2V(\frac{n}{2})$

2.1.4 Heapsort

Definition 2.1: Heap Bedingung

Für jedes Element des Heaps gilt die Beziehung Vater $\geq$ Sohn, beim Entfernen eines Werts muss die Heap-Bedingung wiederhergestellt werden. Diese Prozedur nennt sich **heapify**.

Sortierung

1. **Phase I**: aus Zahlenfolge wird Heap aufgebaut $\mathcal{O}n$
2. **Phase II**: der Heap wird wieder abgebaut und dabei jedes Mal die Wurzel entfernt und durch einen anderen Knoten ersetzt. Danach wird die Heap-Eigenschaft wiederhergestellt (Heapify) $\mathcal{O}(n \log n)$

2.1.5 Vergleich der Laufzeiten

Komplexität	**BubbleSort**	**MergeSort**	**QuickSort**	**HeapSort**
worst-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$
average-case	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Speicherplatz	$\Theta(1)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(1)$

Tabelle 1: Vergleich der Laufzeiten von Sortieralgorithmen

2.2 Selektionsalgorithmen

Definition 2.2: Selektionsalgorithmus

Gegeben sei eine Folge von Zahlen $a[1], \ldots, a[n]$. Ein Selektionsalgorithmus sucht das i-kleinste Element dieser Folge (**Rang** = i). Spezialfälle: Maximum ($i = n$), Minimum ($i = 1$) & Median ($i = \frac{n}{2}$)

2.2.1 Naiver Algorithmus für Minimum und Maximum

- Elemente $a[i]$ und $a[i+1]$ werden paarweise verglichen für $i = 1, \ldots, n-1$
- Dies ergibt die obere Schranke $V(n) \leq 2(n-1)$

2.2.2 Schranken für Minimum und Maximum

Die obere Schranke und untere Schranke für das Suchen des maximalen **oder** minimalen Elements beträgt $\Theta(n)$

Für das Suchen des kleinsten **und** maximalen Elements gilt:

- Durchführen von $\frac{n}{2}$ paarweisen Vergleichen zwischen $a[2i]$ und $a[2i+1]$
- Das Maximum und Minimum lässt sich jeweils durch $\frac{n}{2} - 1$ Vergleichen bestimmen
- Damit gilt $V(n) \leq \frac{3n}{2} - 2$

2.2.3 Random Select

- Quick-Select Algorithmus, bei dem das Pivot zufällig gewählt wird
- $k = q$: Das Pivot ist das gesuchte Element
- $k < q$: Das gesuchte Element liegt im linken Teil
- $k > q$: Das gesuchte Element liegt im rechten Teil

2.2.4 Median of Medians Algorithmus

- Aufteilen der Folge in Gruppen mit fester Länge
- Sortierung der einzelnen Gruppen
- Aus den einzelnen Gruppen werden Mediane bestimmt
- Der Median der Zahlenfolge entspricht dem Median der Mediane der einzelnen Gruppen

Rekursionsgleichung

$$V(n) \leq \frac{7n}{5} + V(\frac{n}{5}) + \frac{4n}{10} + V(\frac{7n}{10})$$

Nach dem Master-Theorem ergibt sich die Laufzeit $\Theta(n)$

2.2.5 Vergleich der Laufzeiten

Komplexität	Random Select	Median of Medians
worst-case	$\Theta(n^2)$	$\Theta(n)$
average-case	$\Theta(n)$	$\Theta(n)$
Speicherplatz	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Tabelle 2: Vergleich der Laufzeiten von Selektionsalgorithmen

3 Hashing

3.1 Grundlegende Methoden

Divisions/- Kongruenzmethode

$h(s) = s \mod m$

Multiplikationsmethode

$h(s) = \lfloor m \cdot ((s \cdot \alpha) \mod 1) \rfloor$

3.2 Wichtige Abschätzung

$$\prod_{i=1}^{n-1} \frac{m-i}{m} = \prod_{i=1}^{n-1} (1 - \frac{i}{m}) \approx \prod_{i=1}^{n-1} e^{-\frac{i}{m}} = e^{-\sum_{i=1}^{n-1} \frac{i}{m}} = e^{-\frac{n(n-1)}{2m}} \approx e^{-\frac{(n-\frac{1}{2})^2}{2m}}$$

3.3 Hashing Verfahren

3.3.1 Hashing mit Verkettung

- Bei Kollision werden Elemente in einer Liste gespeichert
- Belegungsfaktor $\beta = \frac{n}{m} > 1$ möglich

3.3.2 Open Hashing

- Im Kollisionsfall wird ein alternativer freier Platz in der Hashtabelle gesucht
- Löschen stellt ein Problem dar, da die Sondierreihenfolge für andere Schlüssel unterbrochen werden kann
- Unterscheidung zwischen gelöschten und freien Plätzen, sodass beim Suchen nicht fälschlicherweise gemeldet wird, dass der Schlüssel nicht in der Hashtabelle vorhanden ist

- Beim Einfügen werden gelöschte Plätze wie freie Plätze behandelt

Lineares Sondieren

- **Kollisionsfall**: Weitersondierung mit $(k+1) \mod m$
- **Clusterbildung**: zusammenhängende Abschnitte in der Hashtabelle

Double Hashing

- **Kollisionsfall**: Zweite Hashfunktion h' bestimmt die Schrittweite, mit der weitersondiert wird

3.4 Bloomfilter

- Datenstruktur zur Bestimmung, ob ein Element bereits gespeichert wurde
- Beim Einfügen wird an der Stelle des Hashes der Wert auf true gesetzt
- False Positives möglich, d.h. Bloomfilter gibt true zurück, obwohl Element nie gespeichert wurde

4 Dynamisches Programmieren

Definition 4.1: Dynamische Programmierung

Algorithmische Technik zur Lösung von Optimierungsproblemen durch die Zerlegung in einfachere Teilprobleme. Dabei wird jedes Teilproblem nur genau einmal gelöst und dessen Ergebnis in einer Tabelle gespeichert, um es später wiederzuverwenden, statt es neu zu berechnen. Oft werden die einzelnen Teilprobleme bottom-up gelöst.

Definition 4.2: Bellmansches Optimalitätsprinzip

Eine optimale Gesamtlösung besteht aus optimalen Teillösungen der Teilprobleme.

4.1 Matrizen-Kettenmultiplikation

Suchen einer optimalen Klammerung bei der Berechnung des Produkts mehrerer Matrizen

$$m(i,j) = \begin{cases} 0, & i = j \\ \min_{i \le k < j}(m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j), & i < j \end{cases}$$

4.2 Editierdistanz

Elementare Operationen

- Einfügen
- Löschen
- Ersetzen

Levenshtein Distance

$$d(i,j) = \min \begin{cases} d(i-1,j) + 1 \\ d(i,j-1) + 1 \\ d(i-1,j-1) + \delta(a_i, b_j) \end{cases} \qquad \delta(a_i, b_j) = \begin{cases} 1, & a_i \neq b_j \\ 0, & a_i = b_j \end{cases}$$

4.3 Traveling Salesman Person

- Weg über alle Knoten mit niedrigsten Summe an Kantengewichten
- Setzt voraus, dass der Graph einen Hamilton-Kreis enthält

$$g(i,S) = \min_{j \in S}\{m_{i,j} + g(j, s \backslash \{j\})\}$$

Die Laufzeit ergibt sich zu:

$$\mathcal{O}(\text{Größe der Tabelle} \cdot \text{Aufwand pro Eintrag}) \cdot \mathcal{O}(n) = \mathcal{O}(n^2 2^n)$$

4.3.1 0/1 Rucksack Problem

Gesuch: 0/1-Vektor $(a_1, \ldots, a_n) \in \{0,1\}^n$ mit

$$\sum_{i=1}^{n} a_i g_i \le G \text{ und } \sum_{i=1}^{n} a_i v_i \to max$$

$$w(i,h) = \begin{cases} 0, & i = 0 \\ w(i-1,h), & i > 0, h < g_i \text{ (Objekt ist zu groß)} \\ \max\{w(i-1,h), w(i-1,h-g_i) + v_i\}, & \text{sonst} \end{cases}$$

5 Greedy Algorithmen

5.1 Matroide

Sei E eine endliche Menge und $\mathcal{U}$ eine Menge von Teilmengen von E. Die algebraische Struktur $(E,\mathcal{U})$ heißt **Teilmengensystem**, falls:

Leere Menge : $\mathcal{O} \in \mathcal{U}$

Abgeschlossenheit : $A \subseteq B, B \in \mathcal{U} \implies A \in \mathcal{U}$

Damit das Teilmengensystem $(E,\mathcal{U})$ als Matroid bezeichnet werden kann, gilt zusätzlich:

Austauscheigenschaft $A, B \in \mathcal{U}, |A| < |B| \implies \exists x \in B \backslash A : A \cup \{x\} \in \mathcal{U}$

Der kanonische Greedy-Algorithmus liefert für das zugehörige Optimierungsproblem eine optimale Lösung genau dann wenn $(E,\mathcal{U})$ ein Matroid ist.

5.2 Bruchteil-Rucksackproblem

1. Sortiere Objekte nach Wert pro Gewicht: $\frac{v_1}{g_1} \geq \ldots \geq \frac{v_n}{g_n}$
2. Die optimale Lösung hat die Form: $1, \ldots, 1, b, 0, \ldots, 0)$

k := 0
while $(\sum_{i=1}^{k+1} g_i \leq G)$ do k := k+1
b := $\frac{G - \sum_{i=1}^{k} g_i}{g_{k+1}}$ return $(1, \ldots, 1, b, 0, \ldots, 0)$

5.3 Kruskal-Algorithmus

- Algorithmus zur Bestimmung des minimalen bzw. maximalen aufspannendes Baums
- Sortiere Kanten eines ungerichteten Graphen nach Kantengewichten
- Wähle Kanten aus, solange diese keinen Zyklus erzeugen

5.4 Priority Queues und Union-Find

5.4.1 Priority Queues

- Abstrakte Datenstruktur, die folgende Operationen unterstützt
- Konkrete Implementierung: Heap

- **init**: $\mathcal{O}(n)$
- **insert**: $\mathcal{O}(\log n)$
- **extract_max**: $\mathcal{O}(\log n)$

5.4.2 Union-Find

Effiziente Datenstruktur für Greedy-Algorithmen

Make : erstellt ein Baum bestehend aus einer Wurzel mit dem Element

Find : eindeutiger Bezeichner (Repräsentant) wird zurückgeliefert

Union : Mengen werden vereinigt und fortan als eine einzige Menge verwaltet

Pfadkomprimierunng : Elemente werden direkt unter den Repräsentanten verschoben, wird bei jeder Find-Operation aufgerufen

Kruskal-Algorithmus : Effiziente Abfrage, ob das Hinzufügen einer Kante einen Zyklus erzeugt (dabei wird $find(v) =^{?} find(u)$)

5.5 Dijkstra-Algorithmus

- Bestimmt minimalen Abstand eines Knoten zu jedem anderen Knoten

5.6 Huffman-Algorithmus

Definition 5.1: Präfixcode

Kein Codewort darf der Anfang (Präfix) eines anderen Codewortes sein.

Algorithmus zur Bestimmung eines optimalen Präfixcodes zur Datenkompression

Ablauf Jeweils die Buchstaben mit geringsten Aufkommen werden zusammengefasst und jeweils mit dem nächsten über eine Wurzel gestülpt

6 Algorithmen auf Graphen

6.1 Repräsentation von Graphen

Adjazenzliste : Array $A[1..|V|]$ speichert Zeiger auf Liste von Knoten und deren Kanten zu anderen Knoten

Adjazenzmatrix : Darstellung einer Matrix in Matrixform, wobei der i-te Knoten genau dann eine Kante zum j-ten Knoten hat, wenn $D[i,j] \neq \infty$ gilt

6.2 Topologisches Sortieren

- Graph ist ein DAG $\leftrightarrow$ Graph ist topologisch sortierbar
- Knoten ohne Nachfolger werden schrittweise markiert
- Alphabetische Sortierung bei mehreren Vorgängern
- Topologische Sortierung ist umgredrehte Reihenfolge der Knoten

6.3 Floyd-Warshall-Algorithmus

6.4 Ford-Fulkerson-Algorithmus

Bestimmung des maximalen Fluss zwischen Quelle s und Senke t

Kapazitätsbedingung : $f(u,v) \leq c(u,v)$

Symmetriebedingung : $f(u,v) = -f(v,u)$

Kirchhoffsches Gesetz : $\forall u \in V \backslash \{s,t\} : \sum_{v \in V} f(u,v) = 0$

Definition 6.1: Min-Cut-Max-Flow-Theorem

Der maximale Fluss eines Flussdiagrams entspricht der Kapazität des minimalen Schnitt.

Definition 6.2: Edmonds-Karp-Strategie

Als Erweiterungspfad des Ford-Fulkerson-Algorithmus wird immer der kürzeste mögliche gewählt. Dadurch wird die Laufzeit des Ford-Fulkerson-Algorithmus auf $\mathcal{O}(|V| \cdot |E|^2)$ beschränkt.

6.5 Maximales Matching

Definition 6.3: Heiratssatz von Hall

Sei $G = (V,E)$ mit $V = V_1 + V_2$ ein bipartiter Graph. Es gibt ein Matching $M \subseteq E$ mit $|M| = |V_1|$ g.d.w $N(a) \geq |A|$

Merkmal	**Dijkstra-Algorithmus**	**Floyd-Warshall-Algorithmus**
Zielsetzung	**SSSP** (Single-Source Shortest Path): Findet kürzeste Wege von *einem* Startknoten zu allen anderen.	**APSP** (All-Pairs Shortest Path): Findet kürzeste Wege zwischen *allen* Knotenpaaren.
Paradigma	**Greedy-Algorithmus** (S. 71f.): Wählt stets den aktuell erreichbaren Knoten mit der geringsten Distanz.	**Dynamische Programmierung** (S. 87f.): Baut die Lösung schrittweise über Teilprobleme (Zwischenknoten) auf.
Zeitkomplexität	$O(\|V\|^2)$ (naiv) bzw. $O((\|E\| + \|V\|) \log \|V\|)$ bei Verwendung von Heaps (S. 78).	$\Theta(\|V\|^3)$ durch drei geschachtelte Schleifen über alle Knoten (S. 87).
Platzkomplexität	$O(\|V\|)$ für die Speicherung der Distanzliste $l(v)$.	$O(\|V\|^2)$ für die Distanzmatrix $E[i, j]$.
Kantengewichte	Erfordert nicht-negative Gewichte ($w : E \to \mathbb{R}^+$, S. 71).	Kann mit negativen Gewichten umgehen (solange keine negativen Zyklen existieren).
Datenstruktur	Adjazenzliste und Priority Queue (Min-Heap) zur Effizienzsteigerung (S. 77).	Adjazenzmatrix bzw. Distanzmatrix zur Darstellung des Graphen.

Tabelle 3: Vergleich von Dijkstra und Floyd-Warshall gemäß Vorlesungsskript.

7 Algebraische und zahlentheoretische Algorithmen

7.1 Karatsuba und Ofman

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

wobei

- $I = (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$
- $II = (a_{11} + a_{22}) \cdot (b_{11} + b_{22})$

- $III = (a_{11} - a_{21}) \cdot (b_{11} + b_{12})$
- $IV = (a_{11} + a_{12}) \cdot b_{22}$
- $V = a_{11} \cdot (b_{12} - b_{22})$
- $VI = a_{22} \cdot (b_{21} - b_{11})$
- $VII = (a_{21} + a_{22}) \cdot b_{11}$
- $c_{11} = I + II - IV + VI$
- $c_{12} = IV + V$
- $c_{21} = VI + VII$
- $c_{22} = II - III + V - VII$

7.2 Matrizenmultiplikation nach Strassen

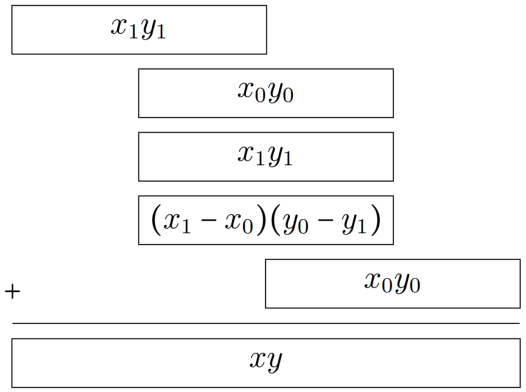

Abbildung 1: Schnelle Matrizenmultiplikation nach Strassen

Laufzeit: $\mathcal{O}(n^{log_2(3)}) \approx \mathcal{O}(n^{1.585})$

7.3 Euklidischer Algorithmus

Aufgrund der Beobachtung $ggt(a, b) = ggt(b, a \mod b)$ folgt der Algorithmus:

```
1 public static int ggt(int a, int b) {
2   while (b != 0) {
3     int temp = b;
4     b = a % b;
```

```
5       a = temp;
6     }
7     return a;
8   }
9
```

Listing 1: Iterativer Euklidischer Algorithmus

7.4 DFT und FFT

7.4.1 Wurzeln komplexer Zahlen

Kartesische Form

$$z = a + i \cdot b$$

Polardarstellung

$$z = r \cdot (\cos\phi + i \cdot \sin\phi)$$

Eulersche Form

$$z = r \cdot e^{i \cdot \phi}$$

7.4.2 Diskrete Fourier-Transformation

$y = V \cdot a$ wobei V die Vandermonde-Matrix ist:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & z & z^2 & \dots & z^{n-1} \\ 1 & z^2 & z^4 & \dots & z^{2(n-1)} \\ 1 & z^3 & z^6 & \dots & z^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z^{n-1} & z^{2(n-1)} & \dots & z^{(n-1)(n-1)} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a^{n-1} \end{pmatrix}$$

Die Inverse V^{-1} einer Vandermonde-Matrix V ist:

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & z & z^2 & \dots & z^{n-1} \\ 1 & z^2 & z^4 & \dots & z^{2(n-1)} \\ 1 & z^3 & z^6 & \dots & z^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z^{n-1} & z^{2(n-1)} & \dots & z^{(n-1)(n-1)} \end{pmatrix}^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \bar{z} & \bar{z}^2 & \dots & \bar{z}^{n-1} \\ 1 & \bar{z}^2 & \bar{z}^4 & \dots & \bar{z}^{2(n-1)} \\ 1 & \bar{z}^3 & \bar{z}^6 & \dots & \bar{z}^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \bar{z}^{n-1} & \bar{z}^{2(n-1)} & \dots & \bar{z}^{(n-1)(n-1)} \end{pmatrix}$$

Zur Bestimmung der Punkt-Wert-Darstellung werden die Koeffizienten eines Polynoms $a_{n-1}x^{n-1} + \ldots + a_0x^0$ mit der Vandermonde-Matrix mit $z = e^{i\frac{2\pi}{n}}$ multipliziert. Der resultierende Vektor y entspricht der Punkt-Wert-Darstellung von a, ausgewertet an den n-ten Einheitswurzeln.

$$\begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix} = V \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Für die Bestimmung des Polynoms $C = A \cdot B$ werden die Punkt-Wert-Darstellungen von A und B multipliziert und die Inverse DFT angewendet:

$$V^{-1} \cdot \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} c_0 \\ \vdots \\ c_{n-1} \end{pmatrix}$$

8 Algorithmen auf Graphen

8.1 Branch and Bound

- Methode zur Lösung von Optimierungsproblemen bei denen keine effizienten Verfahren bekannt sind
- Durchführung einer Verzweigung & Bestimmung einer unteren Schranke
- Beispiel: Traveling Salesman Problem

  Branch

  - Kante (i, j) wird genommen: Setzen der i-ten Zeile und j-ten Spalte & des (j, i)-Eintrags auf ∞ zur Subzyklus-Prävention
  - Kante (i, j) wird nicht genommen: (i, j)-Eintrag wird auf ∞ gesetzt

  Bound Zeilen- und Spaltenreduktion: der minimale Eintrag wird von allen anderen abgezogen